

3장 탐색 알고리즘

목차

◆ 기초적인 탐색 알고리즘

- 순차 탐색
- 이진 탐색
- 이진 트리 탐색

◆ 균형 트리

- 2-3-4 트리
- 레드-블랙 트리
- AVL 트리

◆ 해싱

- 연쇄법
- 선형 탐사법
- 이중 해싱

◆ 기수 탐색

- 디지털 탐색 트리
- 기수 탐색 트라이
- 패트리샤 트리

◆ 외부 탐색

- 인덱스된 순차 접근
- B-트리

탐색(searching)

- ◆ 컴퓨터에 저장된 자료를 신속하고 정확하게 찾아주는 알고리즘
- ◆ 종류
 - 내부탐색 : 주기억 장치에 모두 적재 후 탐색
 - 순차 탐색, 이진 탐색, 이진 트리 탐색, 해싱
 - 외부탐색 : 보조기억 장치의 레코드 탐색
 - 균형 트리 탐색

순차 탐색(sequential searching)

- ◆ 배열에 순서 없이 저장되어 있는 리스트를 탐색
- ◆ 리스트에 있는 첫 번째 레코드부터 시작하여 마지막 레코드까지 차례로 탐색 키를 비교
- ◆ 새로운 레코드는 배열의 마지막에 삽입
- ◆ 비교 횟수
 - 배열로 구현된 순차 탐색은 성공적이지 않은 탐색에 대해 항상 $N + 1$ 회의 비교가 필요하며, 성공적인 탐색에 대해서는 평균적으로 $N/2$ 회의 비교가 필요
 - 정렬된 연결 리스트로 구현된 순차 탐색은 성공적인 탐색과 성공적이지 않은 탐색 모두 평균적으로 $N/2$ 회의 비교가 필요

순차 탐색 알고리즘

```
sequentialSearch(a[], search_key, n)  
  i ← 0;
```



```
end sequentialSearch()
```

순차 탐색의 성능 향상 방법

◆ 접근 빈도수를 알 때

- 빈도수가 가장 높은 레코드를 파일의 처음에, 두 번째 높은 레코드를 2번째에, ... 순으로 저장
- 적은 양의 레코드가 집중적으로 접근될 때 아주 유용함

◆ 접근 빈도수를 모를 때

- “자체-조직(self-organizing)” 탐색
- 레코드가 접근될 때마다 파일의 시작 부분으로 이동하는데, 연결 리스트로 구현하는 것이 적합함
- 많은 접근이 인접한 레코드에 대해 이루어지는 경우에 성능 향상 효과가 있음

이진 탐색(binary search)

◆ 분할-정복 방법을 적용


◆ 성능 특성

- 이진 탐색은 성공적인 탐색과 성공적이지 않은 탐색 모두 $\log N + 1$ 회 이상의 비교를 하지 않음

$$C_N = C_{N/2} + 1$$

- 리스트가 정렬되어 있어야 하므로 리스트를 정렬된 상태로 유지하는데 추가적인 비용 소요

이진 탐색 알고리즘

```
binarySearch(a[], search_key, n)
  left ← 0; right ← n-1;
  while (right ≥ left) do {
    
  }
  return -1; // key 값이 존재하지 않음
end binarySearch()
```


이진 트리 탐색

◆ 이진 탐색 트리

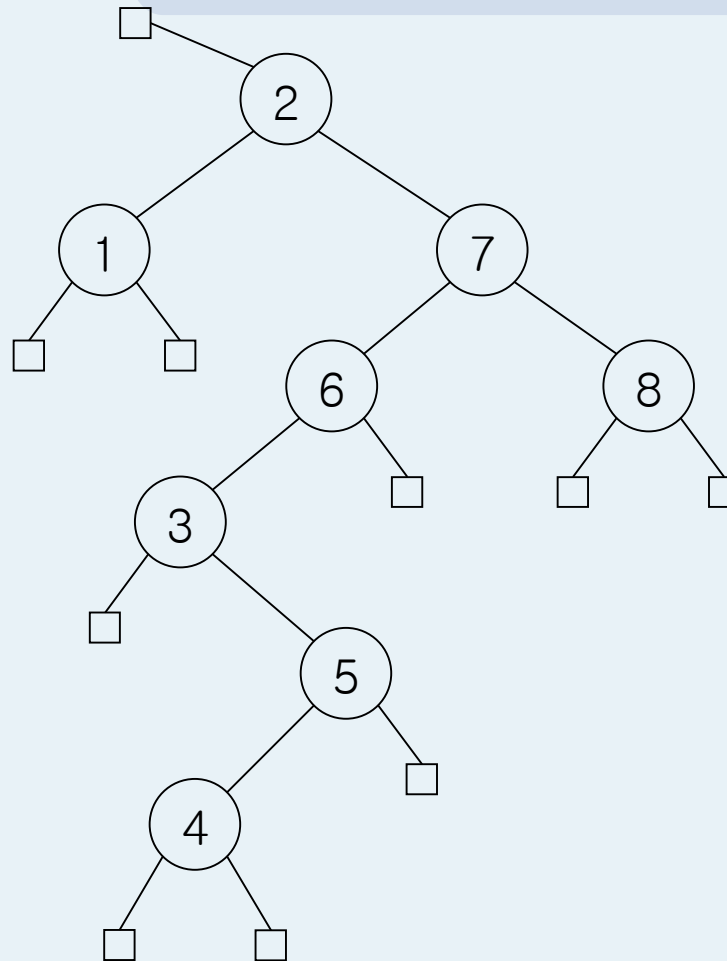
- 작은 키를 가지고 있는 모든 레코드는 왼쪽 부분트리에 있음
- 큰 키를 가지고 있는 모든 레코드는 오른쪽 부분트리에 있음

◆ 탐색 과정

- 루트와 주어진 키를 비교
- 키가 루트보다 작다면, 왼쪽 부분트리로 이동
- 키가 루트와 같다면, 종료
- 키가 루트보다 크다면, 오른쪽 부분트리로 이동

◆ 트리를 중위 순회하면 데이터를 정렬하는 효과가 있음

이진 탐색 트리의 예

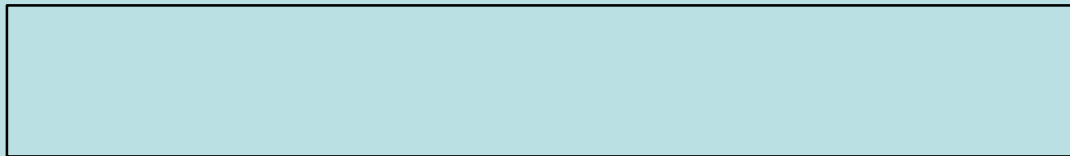


이진 트리 탐색 알고리즘

```
binaryTreeSearch(T, search_key)
```

```
  x ← T;
```

```
  while (x ≠ null) do {
```



```
  }
```

```
  return -1; // 탐색 실패
```

```
end binaryTreeSearch()
```

성능 특성

- ◆ N 개의 임의의 키로 이루어진 이진 탐색 트리를 탐색하고 삽입하는 연산은 평균적으로 $\log N$ 회의 비교 필요
- ◆ 최악의 경우 N 개의 키를 가진 이진 탐색 트리를 탐색하는데 N 회의 비교 필요
 - 키가 정렬되거나 역순으로 정렬된 순서로 삽입
 - 최초 비어 있는 트리에 키들이 A Z B Y C X ... 와 같은 순서로 삽입

균형 트리(balanced tree)

◆ 이진 트리 알고리즘

- 최악의 경우 성능이 나쁨 : 정렬된 화일 또는 역순으로 정렬된 화일, 큰 키와 작은 키가 교대로 나오는 파일

◆ 성능 개선

- 퀵 정렬의 경우 성능 개선 방법은 확률에 의해 임의의 분할 원소를 선택하는 수 밖에 없음
- 이진 트리 탐색의 경우에는 트리를 균형 있게 유지하면 최악의 상황을 피할 수 있음
- 일반적인 개념은 쉽게 기술할 수 있지만, 실제 구현에서는 특수한 상황을 고려해야 함

2-3-4 트리(2-3-4 Tree)

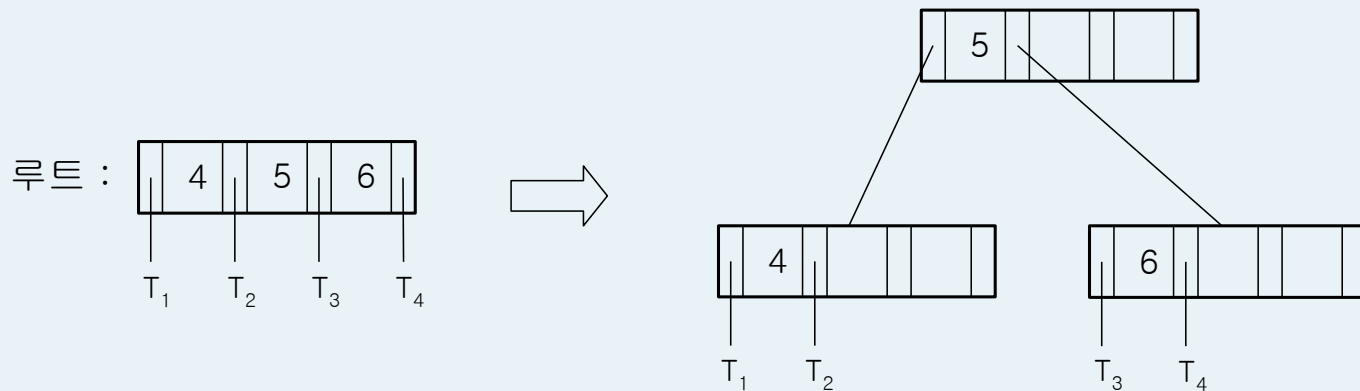
- ◆ 이진 탐색 트리(1개의 키, 2개의 링크)에서 발생하는 최악의 상황을 방지하기 위해 다음과 같이 자료구조를 변경
 - 트리에 있는 하나의 노드가 하나 이상의 키를 가질 수 있음
 - 3-노드(2개의 키, 3개의 링크)와 4-노드(3개의 키, 4개의 링크)를 허용
 - 트리의 균형에 대해 신경 쓰지 않아도 완벽한 균형 트리가 만들어 짐

분할 과정(1)

- ◆ 루트가 4-노드인 경우
 - 세 개의 2-노드로 변환
 - 루트의 분할은 트리의 높이를 하나 증가시킴
- ◆ 부모가 2-노드인 4-노드의 경우
 - 중간 키를 부모로 보내어 3-노드에 연결된 두 개의 2-노드로 변환
- ◆ 부모가 3-노드인 4-노드의 경우
 - 4-노드에 연결된 두 개의 2-노드로 변환

분할 과정 (2)

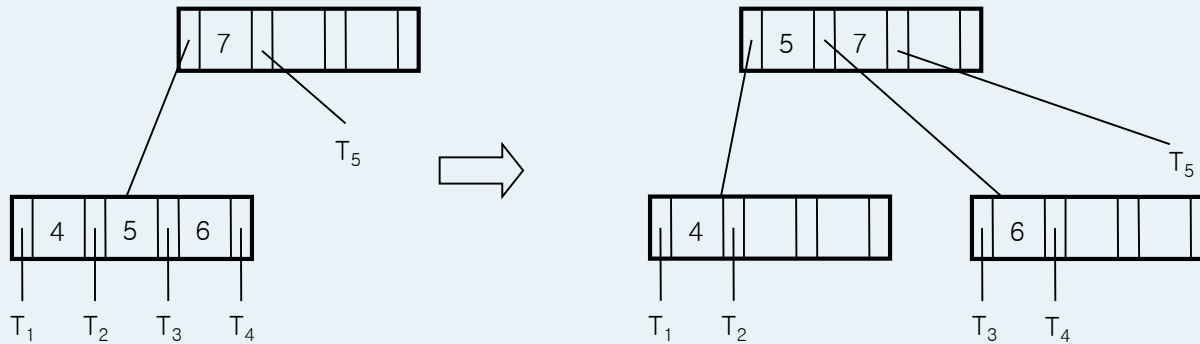
- ◆ 4-노드가 2-3-4 트리의 루트인 경우
 - 노드가 루트인 경우의 분할(T_i 는 서브트리)



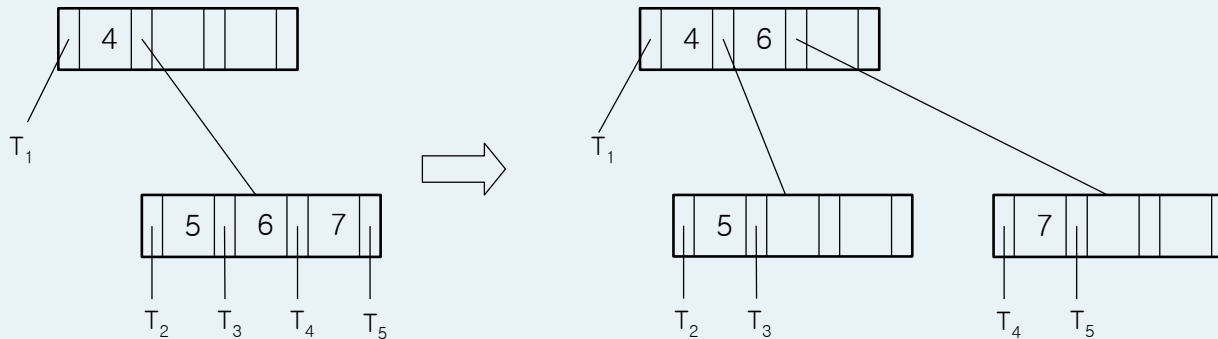
분할 과정(3)

◆ 4-노드가 2-노드의 자식인 경우의 분할

- 4-노드가 2-노드의 왼쪽 자식인 경우



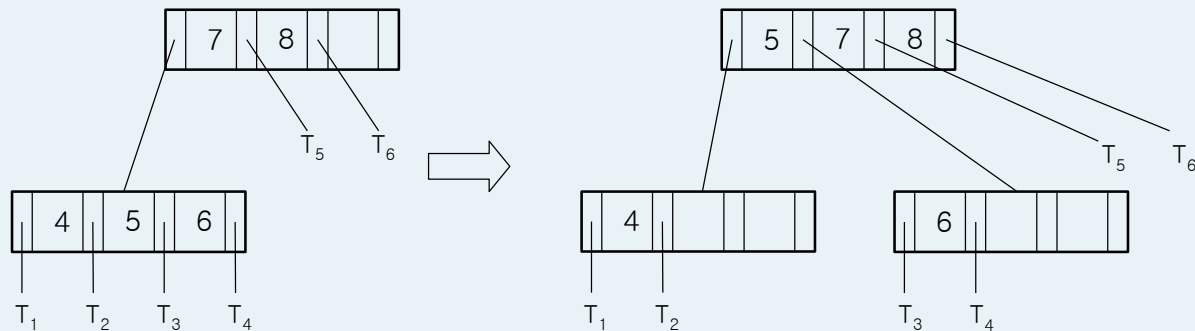
- 4-노드가 2-노드의 왼쪽 중간 자식인 경우



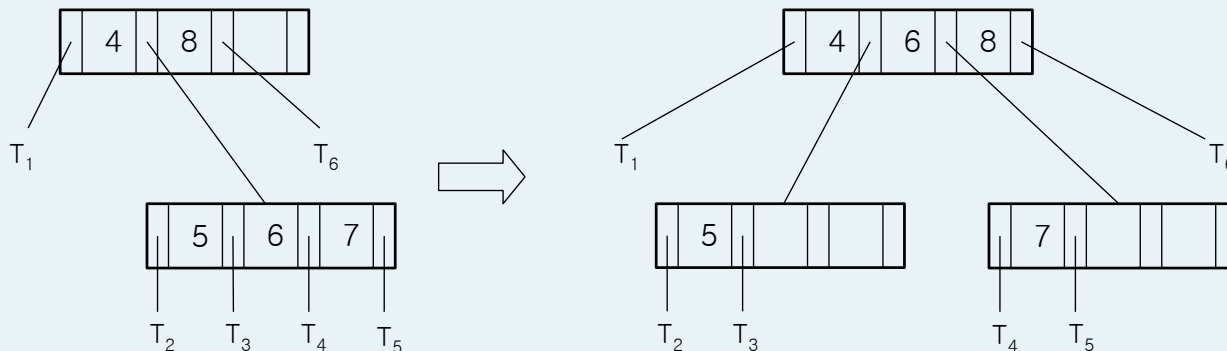
분할 과정(4)

◆ 4-노드가 3-노드의 자식인 경우

➤ 4-노드가 3-노드의 왼쪽 자식인 경우

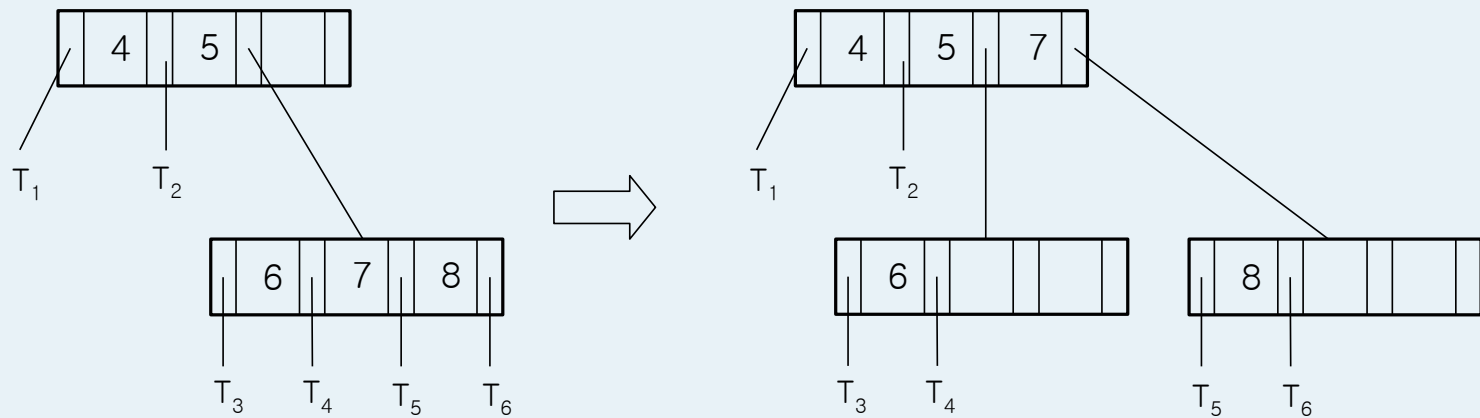


➤ 4-노드가 3-노드의 왼쪽 중간 자식인 경우



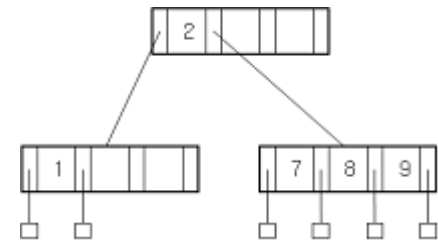
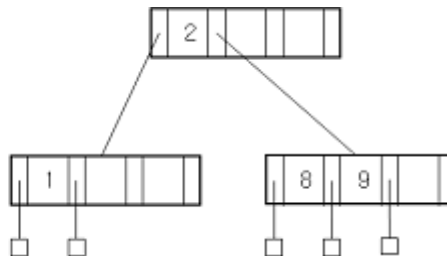
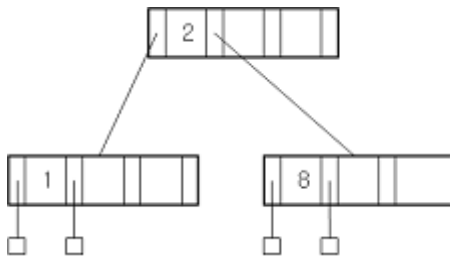
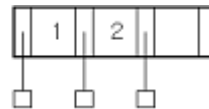
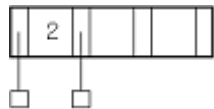
분할 과정(5)

- 4-노드가 3-노드의 오른쪽 자식인 경우



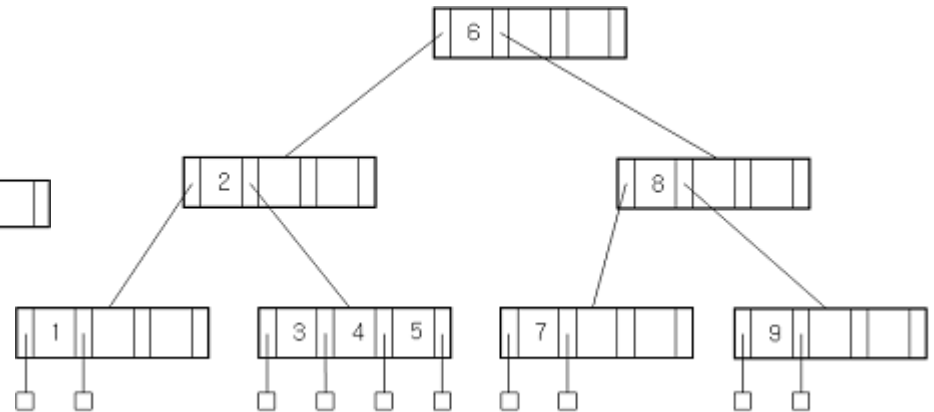
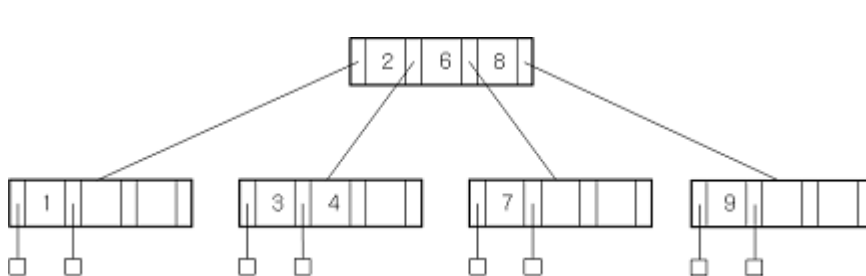
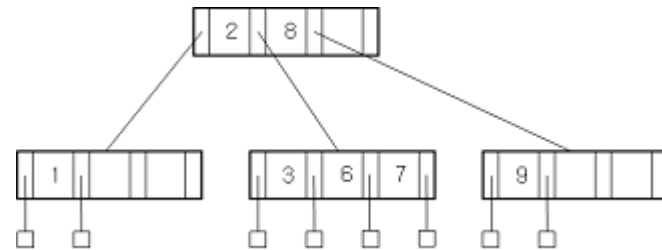
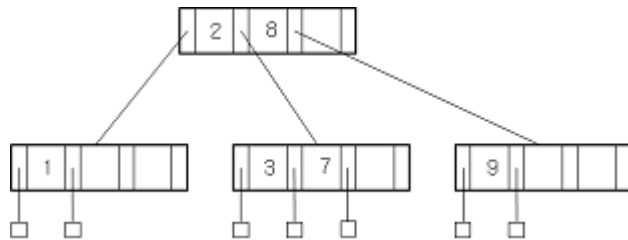
2-3-4 트리의 구축 과정 (1)

키 삽입 (2, 1, 8, 9, 7, 3, 6, 4, 5)



2-3-4 트리의 구축 과정 (2)

키 삽입 (2, 1, 8, 9, 7, 3, 6, 4, 5)

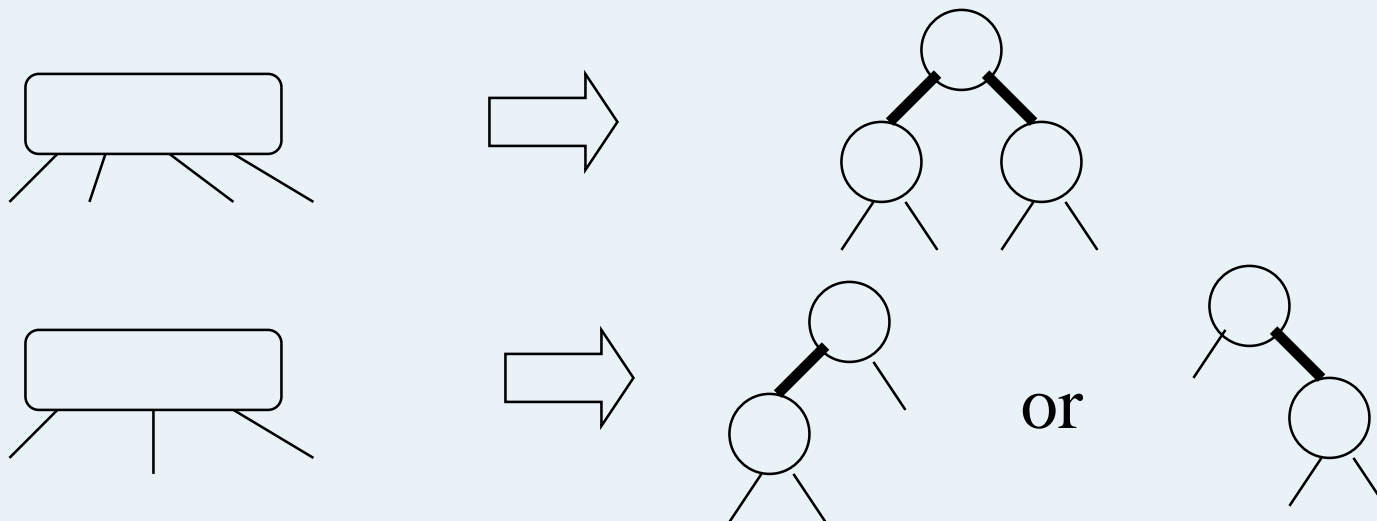


성능 특성

- ◆ N -노드로 이루어진 2-3-4 트리의 탐색은 $\log N + 1$ 개 이상 노드를 방문하지 않음
- ◆ N -노드로 이루어진 2-3-4 트리의 삽입은 최악의 경우 $\log N + 1$ 개의 노드를 분할하며, 평균적으로는 하나 이하의 노드를 분할함
 - 2-3-4 트리에서 4-노드의 개수가 적음
- ◆ 일반적으로 높이가 h 인 2-3-4 트리는 $2^{h+1}-1$ 과 $4^{h+1}-1$ 사이의 키를 포함
- ◆ N 개의 키를 포함하는 2-3-4 트리의 높이는 $\log_4(N+1)-1$ 과 $\log_2(N+1)-1$ 사이

레드-블랙 트리(red-black tree)

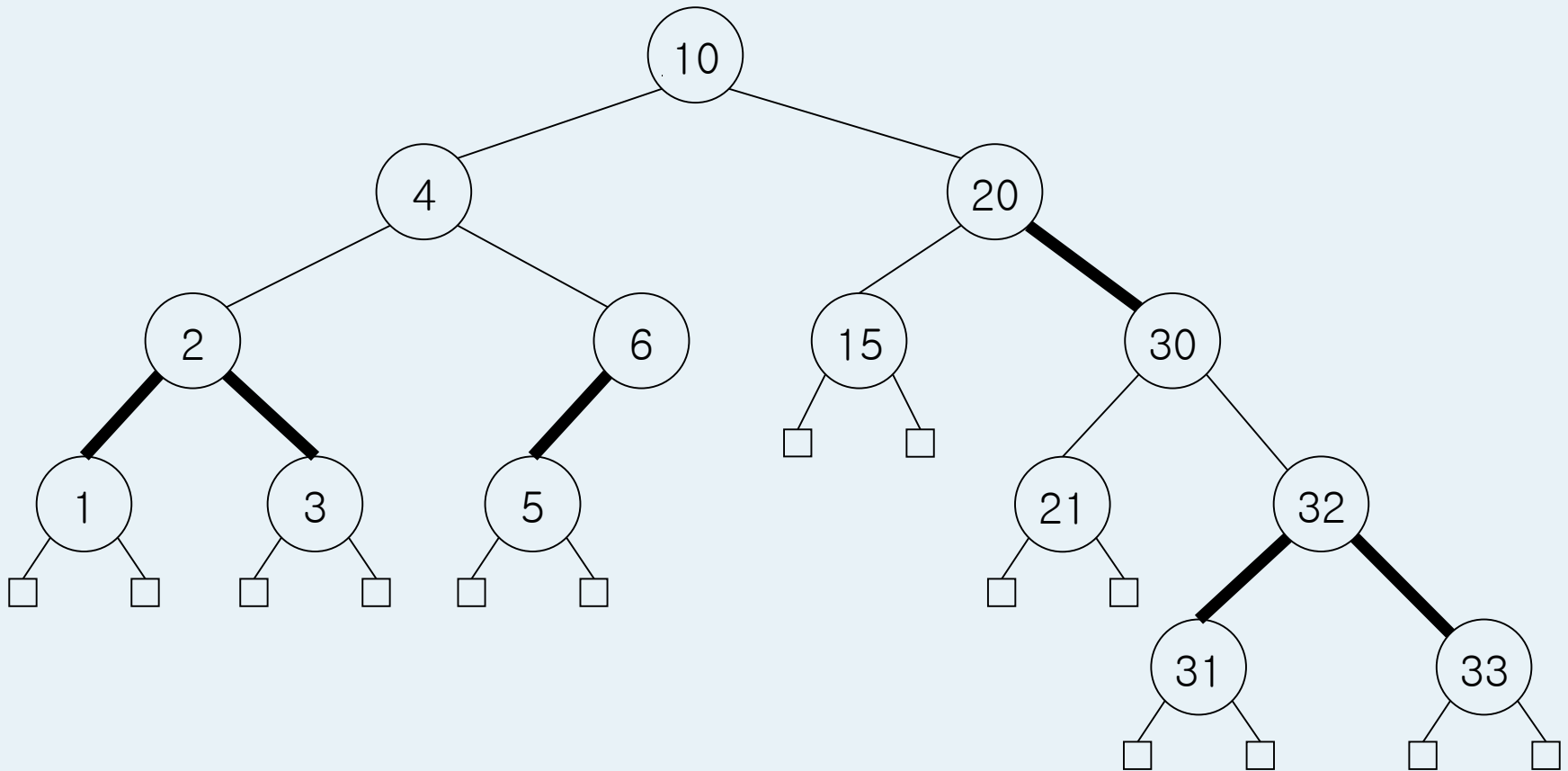
- ◆ 2-3-4 트리를 표준 이진 트리로 표현하는 방법
- ◆ 노드당 추가로 1 비트가 더 필요함
- ◆ 블랙 링크는 보통의 노드이고, 3-노드와 4-노드는 레드 링크로 연결된 이진 트리 표현



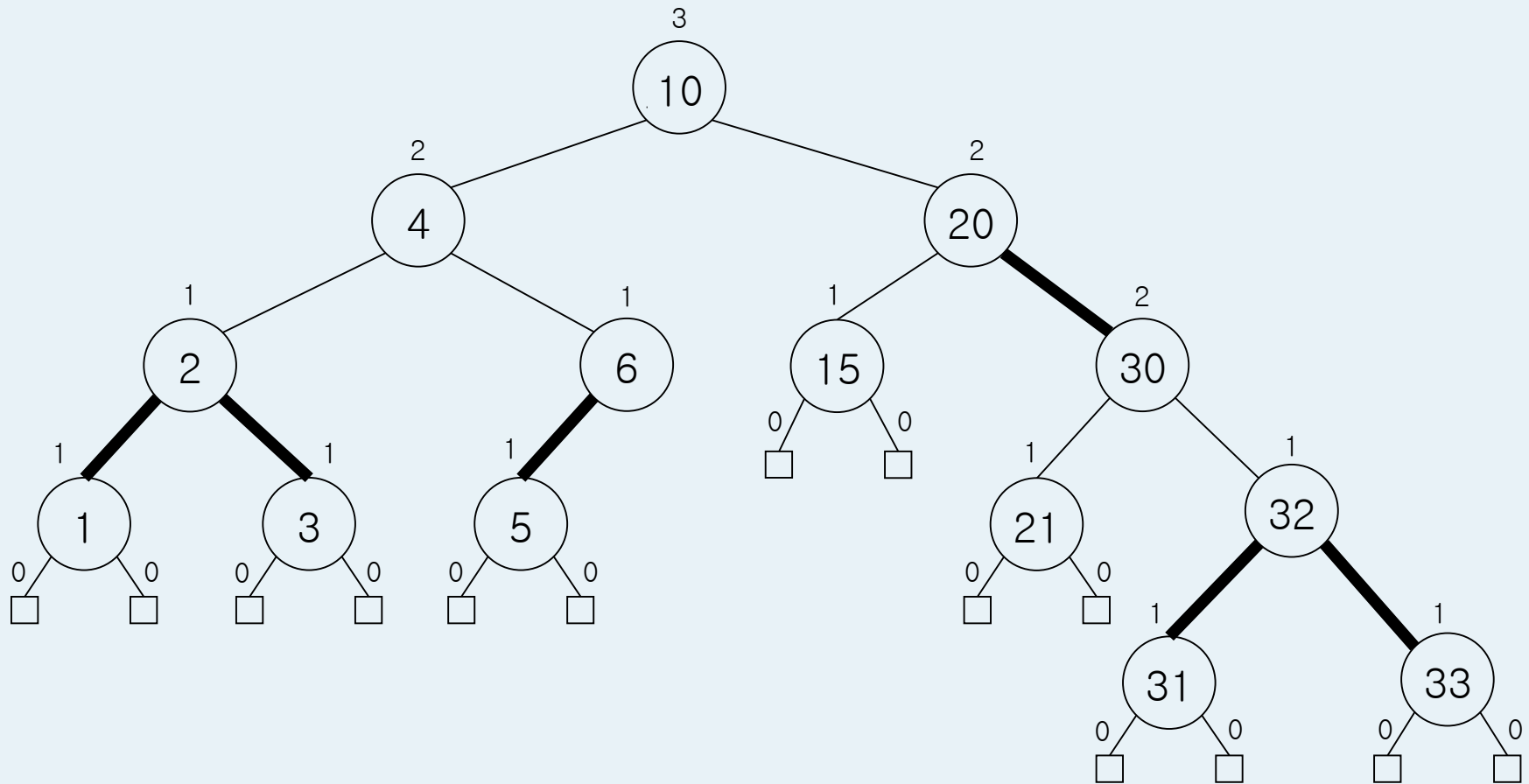
성질

- ◆ 루트나 외부 노드는 모두 블랙
- ◆ 루트에서 외부 노드까지의 경로 상에는 2개의 연속된 레드 노드가 포함되지 않음
- ◆ 루트에서 각 외부 노드까지의 경로에 있는 블랙 노드의 수는 모두 같음
- ◆ 동일한 키를 가지는 레코드는 노드의 왼쪽과 오른쪽에 모두 올 수 있음
- ◆ 동일한 키가 여러 개 있을 때 주어진 키를 갖는 모든 노드를 찾는 것이 어려움

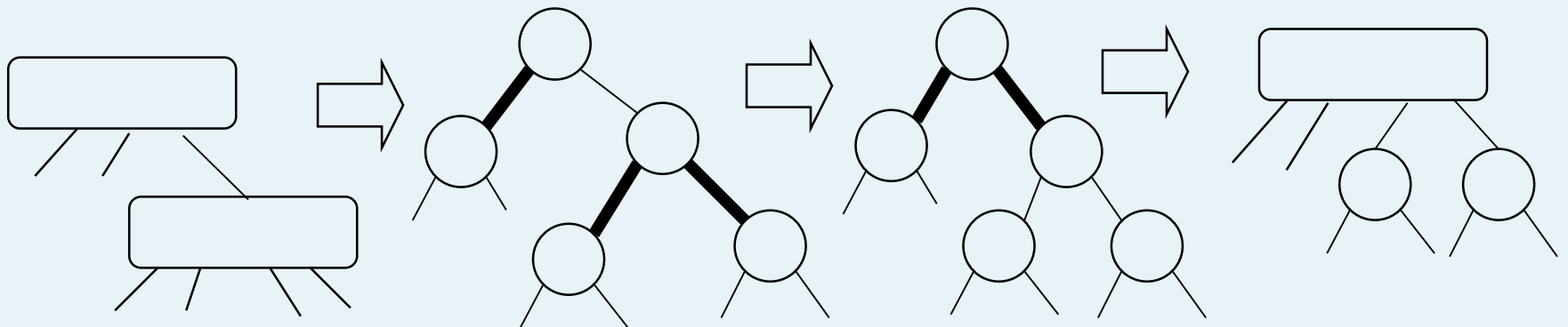
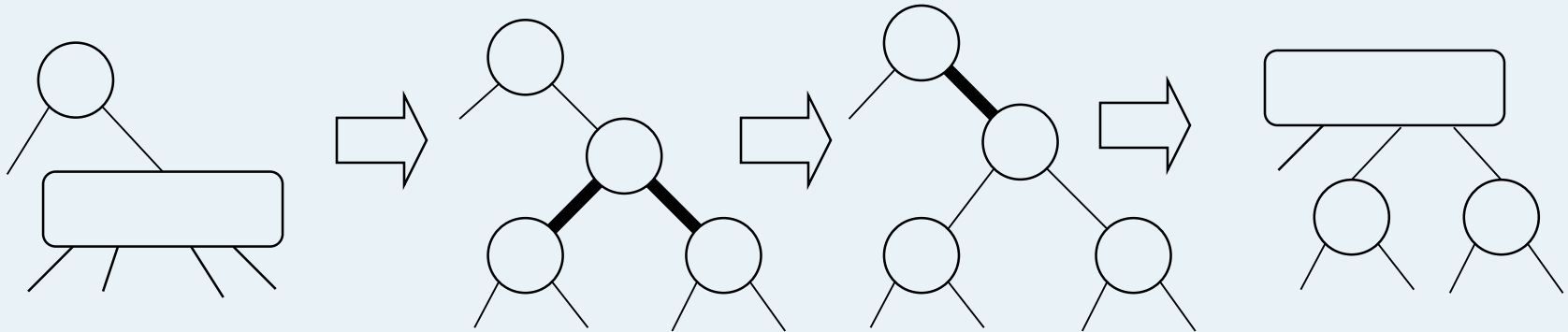
레드-블랙 트리의 예



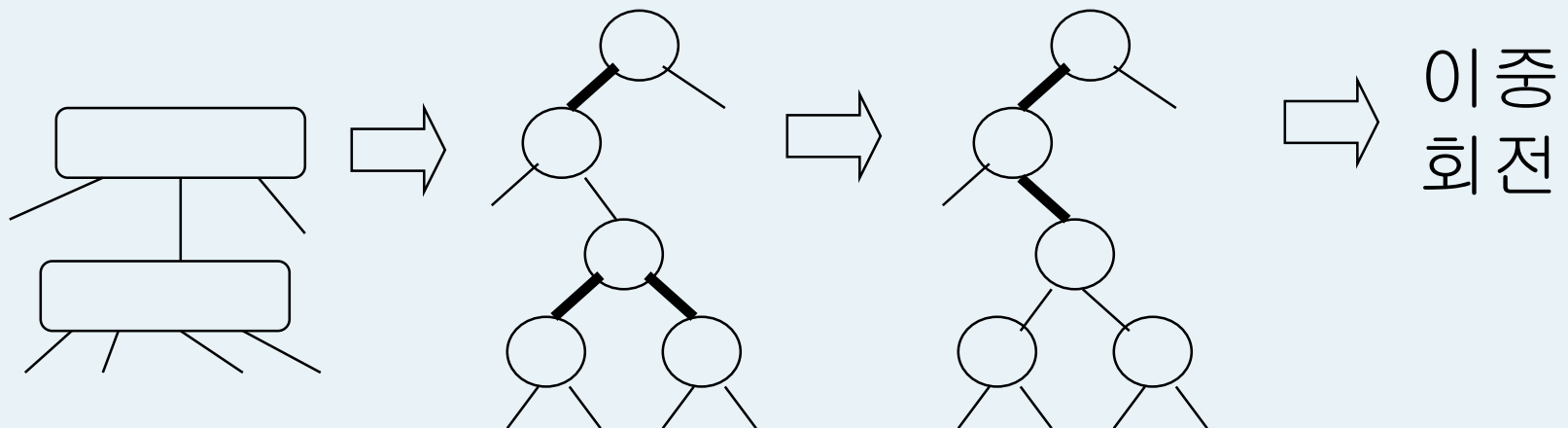
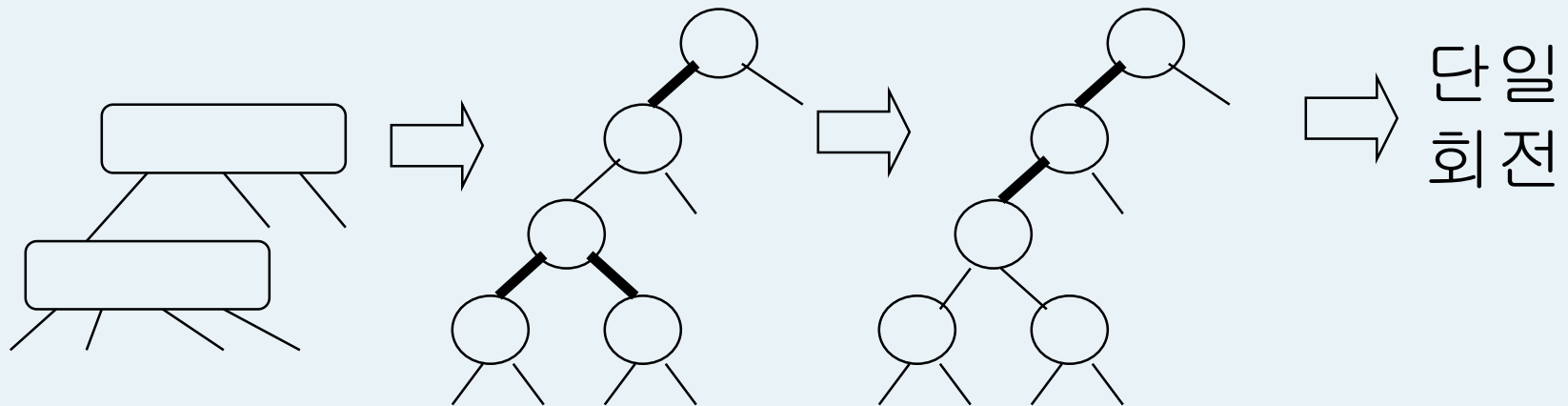
레드-블랙 트리의 서열(rank)



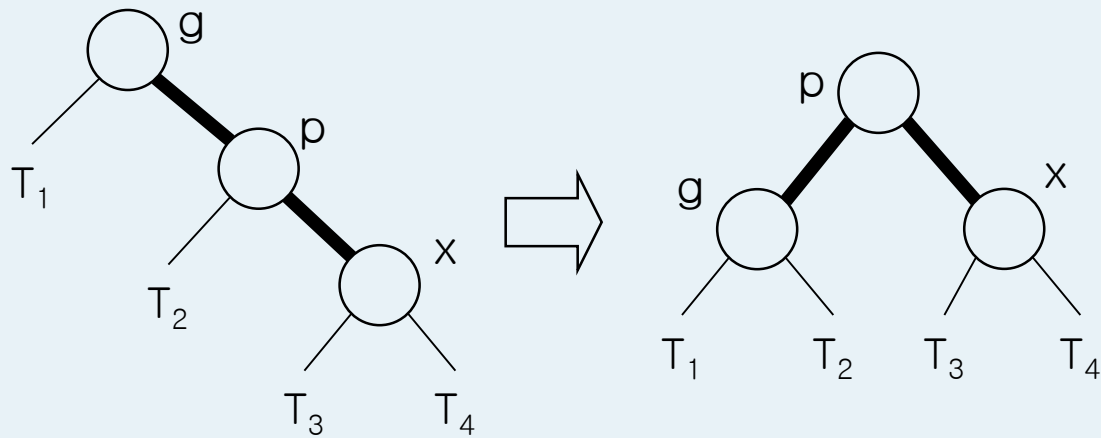
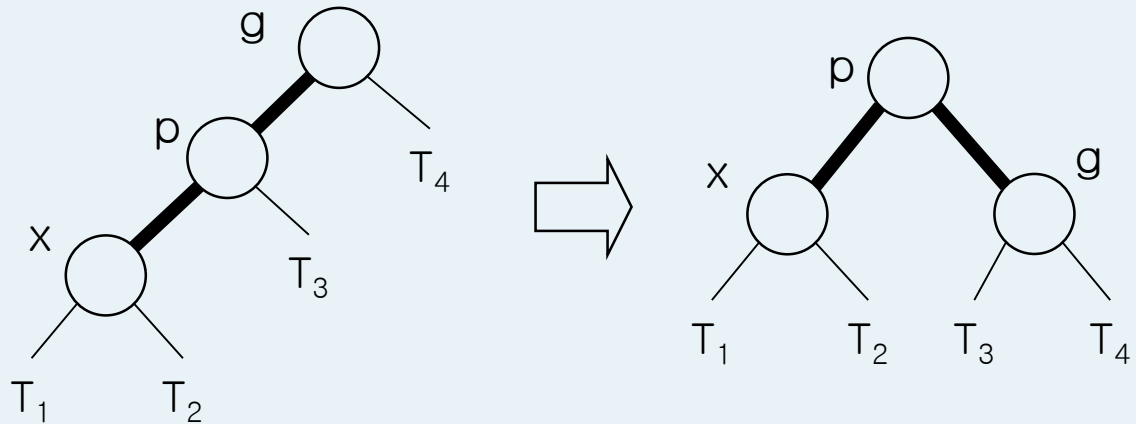
4-노드의 분할



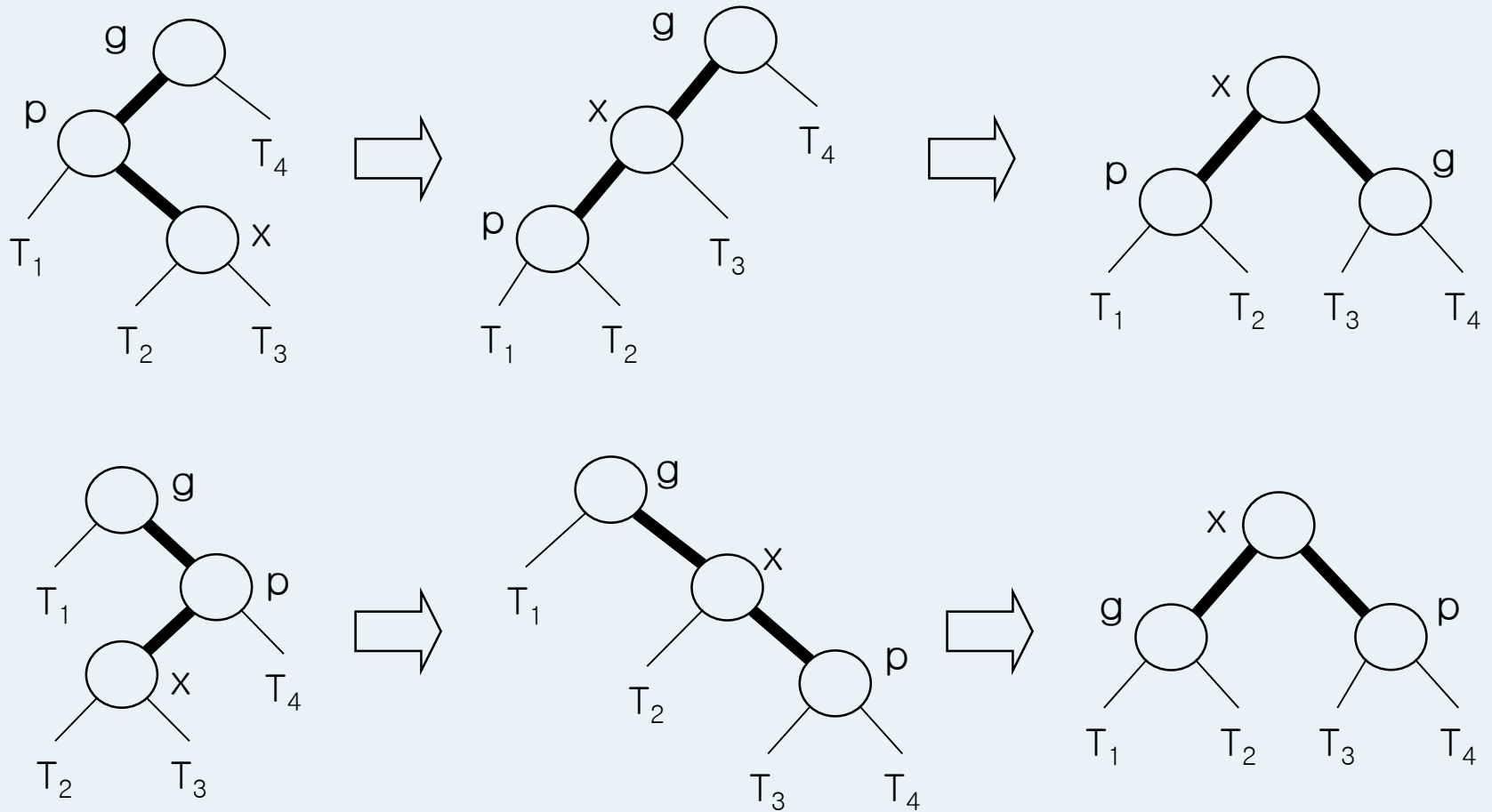
4-노드의 분할(회전 필요)



단일 회전

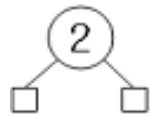


이중 회전

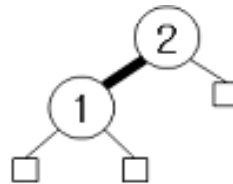


레드 블랙 트리의 구축과정 (1)

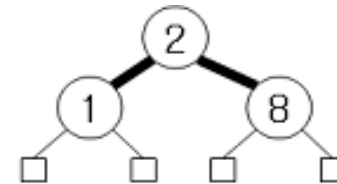
◆ 키 (2, 1, 8, 9, 7, 3, 6, 4, 5)



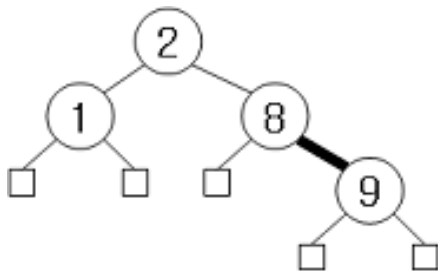
2 삽입



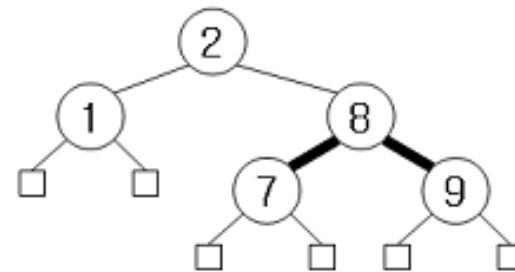
1 삽입



8 삽입



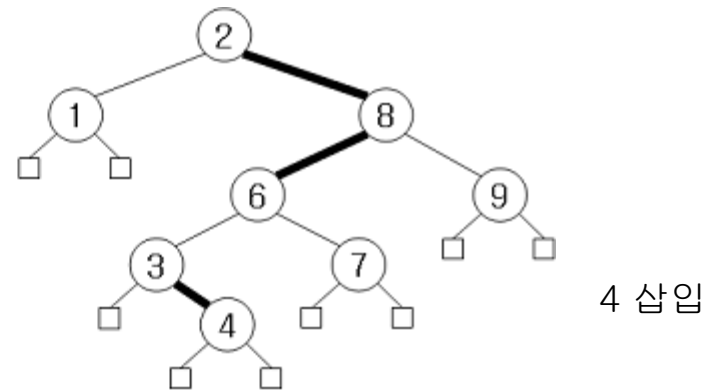
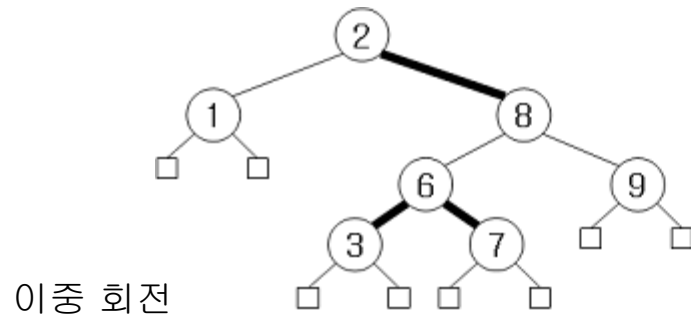
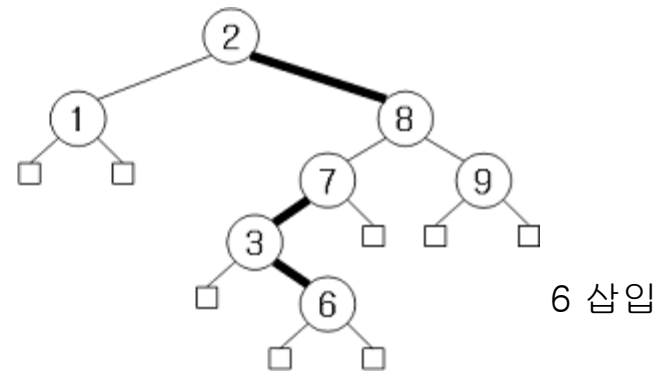
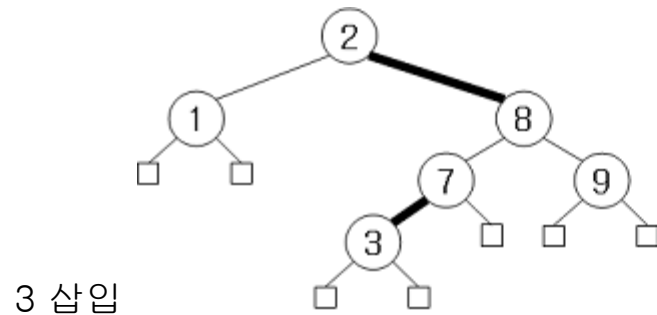
9 삽입



7 삽입

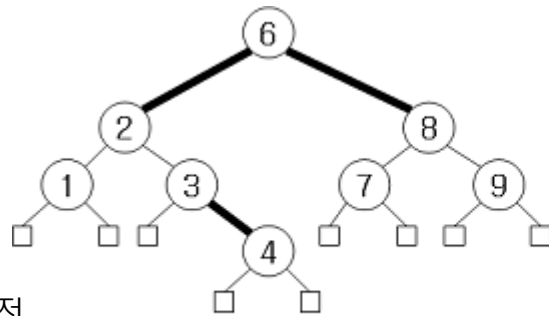
레드 블랙 트리의 구축과정 (2)

◆ 키 (2, 1, 8, 9, 7, 3, 6, 4, 5)

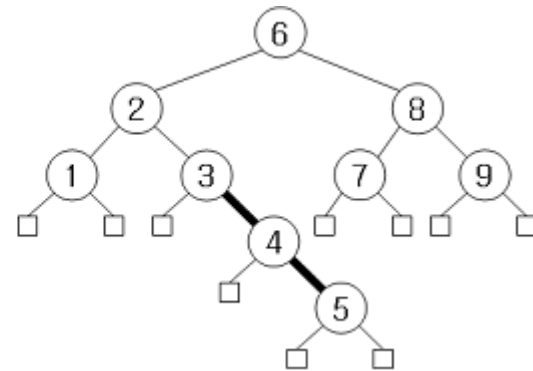


레드 블랙 트리의 구축과정 (3)

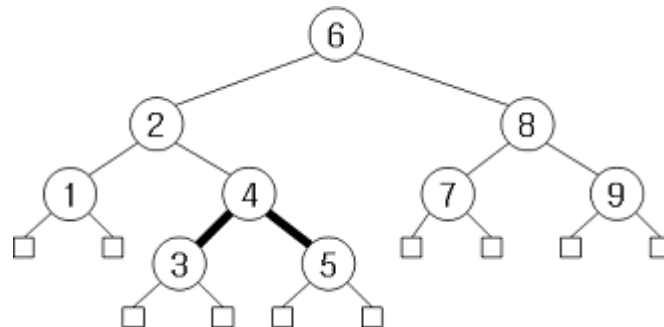
◆ 키 (2, 1, 8, 9, 7, 3, 6, 4, 5)



이중 회전



5 삽입



단일 회전

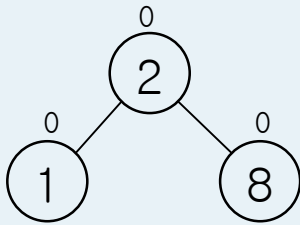
성능 특성

- ◆ r 을 서열(rank)이라 할 때, 노드 수(키 값의 수) N 은 $N \geq 2^r - 1$
- ◆ 레드-블랙 트리의 높이 h 는 $h \leq 2\log(N+1)$ 이므로, 삽입 시간 복잡도는 $O(\log M)$
- ◆ 레드-블랙 트리는 이진 탐색 트리이기 때문에 탐색은 일반 이진 탐색 트리의 탐색 연산으로 수행할 수 있으므로 레드-블랙 트리의 탐색 시간 복잡도는 $O(\log M)$

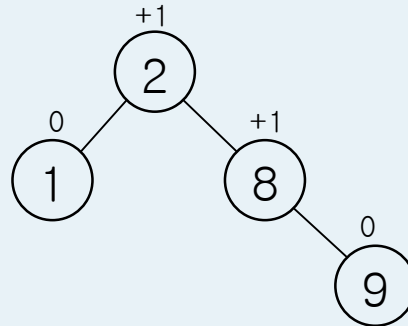
AVL 트리

- ◆ 러시아의 수학자 Adelson-Velskii와 Landis가 고안한 높이 균형 이진 탐색 트리(height-balanced binary search tree)
- ◆ 오른쪽 서브트리와 왼쪽 서브트리의 높이 차이가 2 이상이 되면 회전을 통해 트리의 높이 차를 1 이하로 유지
- ◆ 높이 차 = 오른쪽 서브트리의 높이 - 왼쪽 서브트리의 높이

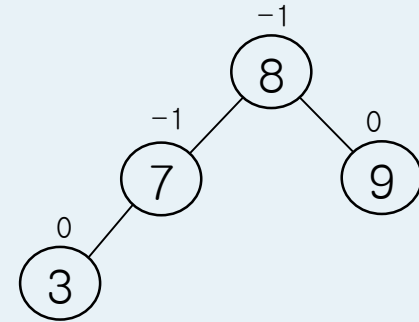
AVL 트리의 높이 차



높이 차가
0인 경우



높이 차가
양수인 경우

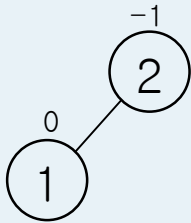


높이 차가
음수인 경우

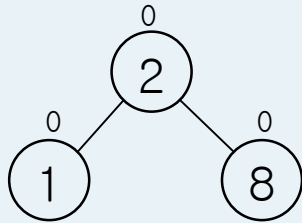
AVL 트리의 구축 과정(1)



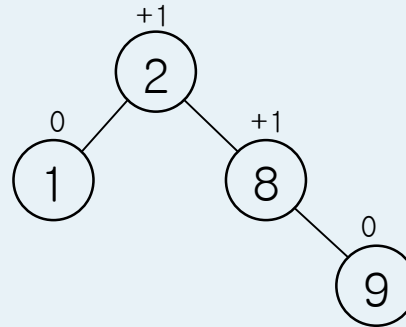
2 삽입



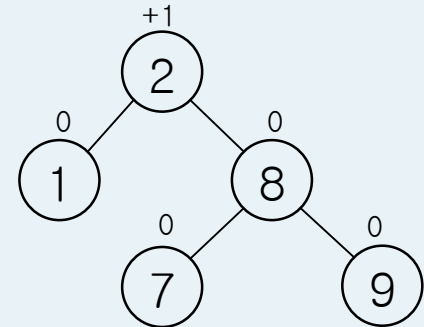
1 삽입



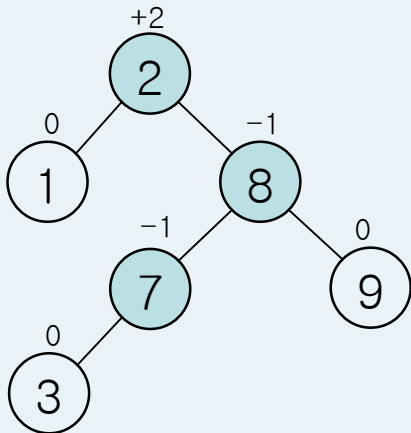
8 삽입



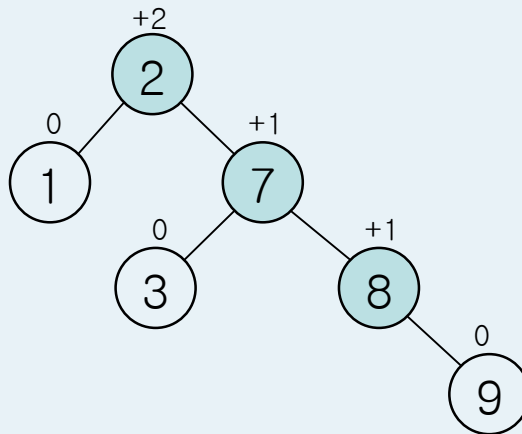
9 삽입



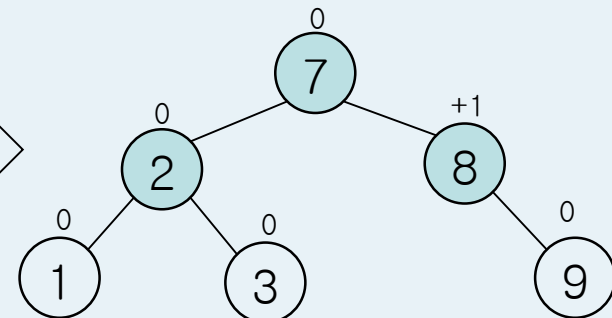
7 삽입



RL

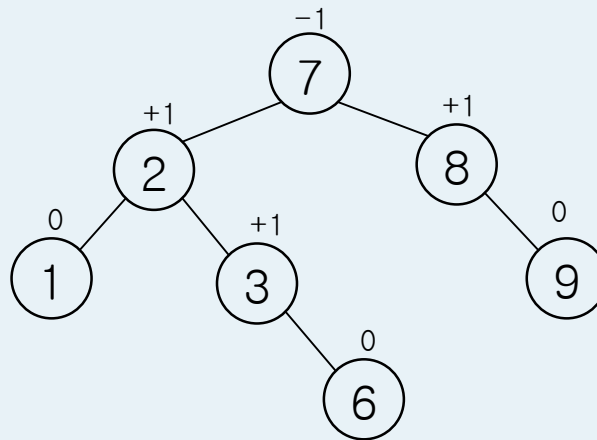


RR



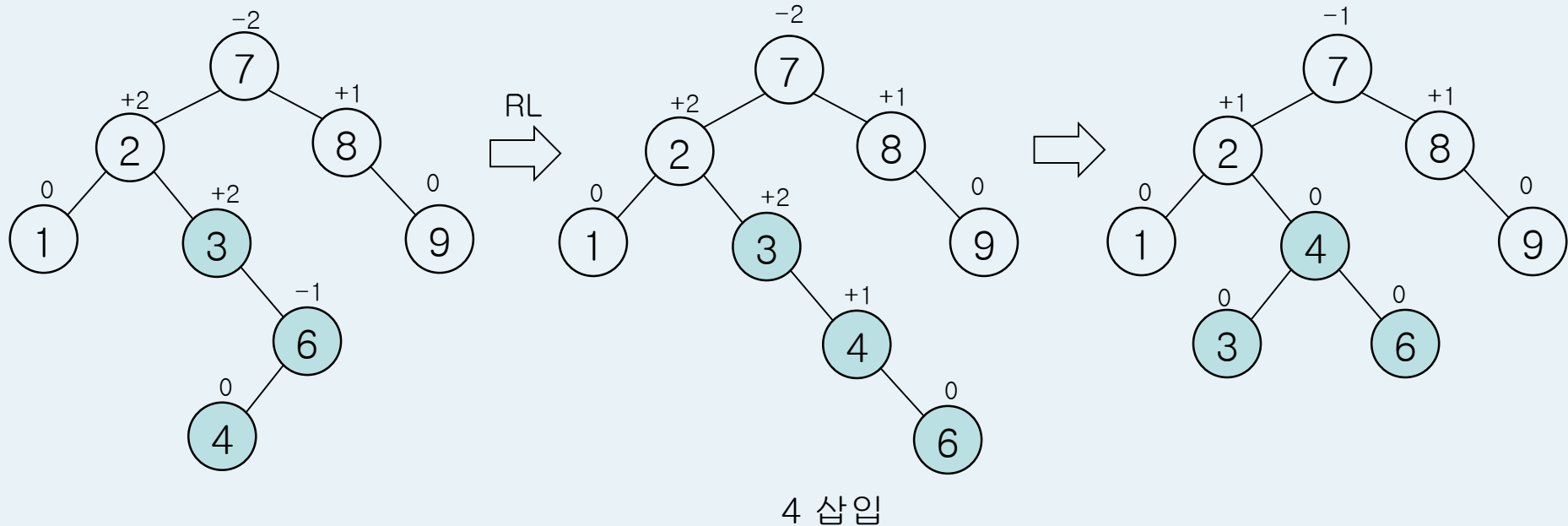
3 삽입

AVL 트리의 구축 과정(2)

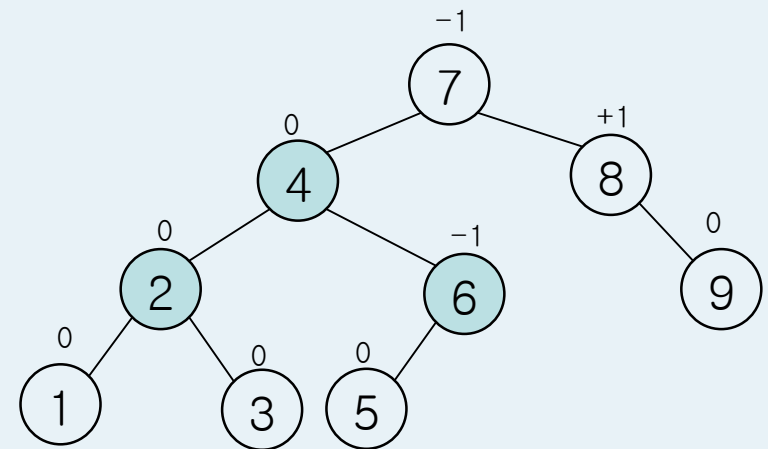
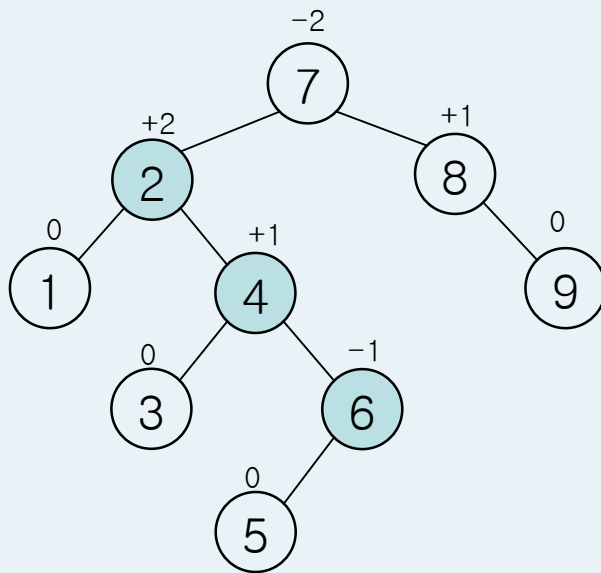


6 삽입

AVL 트리의 구축 과정(3)



AVL 트리의 구축 과정(4)



5 삽입

해싱(hashing)

- ◆ 모든 키의 레코드를 산술 연산에 의해 한번에 바로 접근할 수 있는 기법
- ◆ 해싱의 단계
 - 해시 함수(hash function)를 통해 탐색 키를 해시 테이블 주소로 변환
 - 같은 테이블 주소로 사상되었을 경우에는 충돌을 해결(collision-resolution)해야 함

특징

◆ 시간과 공간의 균형

- 메모리의 제한이 없을 경우 : 키를 메모리 주소로 사용하면 어떤 탐색이든지 한 번의 메모리 접근으로 수행 가능
- 시간에 제한이 없을 경우 : 최소한의 메모리를 사용하여 데이터를 저장하고 순차 탐색

◆ 해싱은 이러한 두 극단 사이에서 합리적인 균형을 이룰 수 있는 방법을 제공

◆ 해싱을 사용하는 목적은 가용한 메모리를 효과적으로 사용하면서 빠르게 메모리에 접근하기 위함

해시 함수(1)

- ◆ 키의 전체 집합을 U 라 하고, 해시 테이블을 $\pi[0..m-1]$ 이라 할 때, 해시 함수 h 는 다음과 같이 키값을 테이블 주소로 변환하는 함수임

$$h: U \rightarrow \{0, 1, \dots, m-1\}$$

해시 함수(2)

◆ 나눗셈법

- 다음과 같은 해시 함수 사용

$$h(k): k \bmod m$$

- m 의 값은 주의해서 선택해야 함
 - 만약 m 을 2의 거듭제곱, 예를 들어 2^r 으로 택한다면 $h(k)$ 는 k 의 하위 r 비트의 값으로 됨
 - 키가 십진수로 표현된 경우에도 10의 거듭제곱이 아닌 m 을 선택
- 따라서, m 은 2의 거듭제곱과 상당한 차이가 있는 소수를 선택하는 것이 좋음

해시 함수(3)

◆ 곱셈법

- 소수가 아닌 m 을 택해야 하는 경우 사용
- m 의 값이 중요하지 않음
- 계산을 간단하게 하기 위해 보통 m 은 2의 거듭제곱, 즉 $m=2^p$ 로 선택

◆ 유니버설 해싱

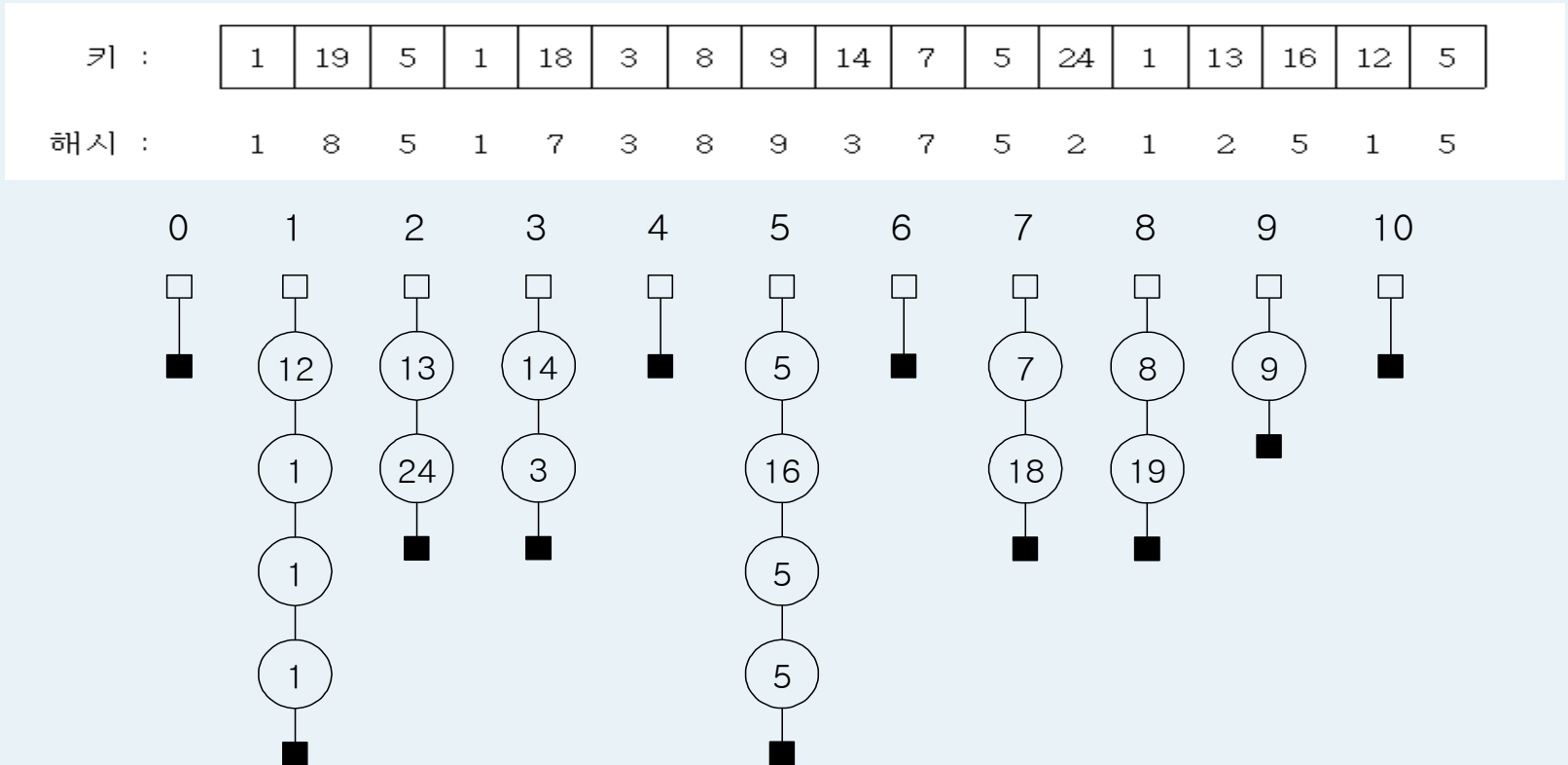
- 해시 함수마다 아주 나쁜 성능을 보이는 입력 데이터의 패턴이 존재
- 실행시에 해시 함수의 큰 집합으로부터 임의의 해시 함수를 선택하도록 하여 해시 함수의 편향된 사용이 불가능하여 악의적인 사용자를 배제함

연쇄법(chaining)

- ◆ 동일 주소로 해시되는 모든 원소가 연결 리스트 형태로 연결됨
- ◆ 장점
 - 원소의 삭제가 용이
- ◆ 단점
 - 포인터 저장을 위한 기억공간이 필요
 - 기억장소 할당이 동적으로 이루어져야 함

해시 함수의 예

$h(k): k \bmod 11$



성능 특성

- ◆ 성공적이지 않은 탐색을 위한 리스트의 평균 거리
 - $(1+5+3+3+1+5+1+3+3 + 2+1)/11 \approx 2.55$ 번
- ◆ 성공적인 탐색을 위한 리스트의 평균 거리
 - $(7 \cdot 1 + 6 \cdot 2 + 2 \cdot 3 + 2 \cdot 4)/17 \approx 1.94$ 번
- ◆ 연쇄법은 m 개의 추가 기억장소가 사용되지만, 순차 탐색에 비해 평균적으로 해시 테이블의 크기 m 에 비례하여 비교 횟수를 줄임

선형 탐사법(linear probing)

◆ 해시 함수

$$h(k) = k \bmod m$$

- 개방 주소법(open addressing) 사용 : m 값이 입력 원소의 수보다 커야 함

◆ 클러스터링이 발생

- 점유된 위치가 연속적으로 나타나는 뭉치가 있으면 이것이 점점 더 커지는 현상
- 이러한 뭉치는 평균 탐색 시간을 증가시킴

선형 탐사법의 수행과정(1)

$h(k): k \bmod 19$

키 :

1	19	5	1	18	3	8	9	14	7	5	24	1	13	16	12	5
---	----	---	---	----	---	---	---	----	---	---	----	---	----	----	----	---

해시 : 1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]
	1																	
19	1																	
19	1				5													
19	1	1			5													
19	1	1			5													18
19	1	1	3		5													18

선형 탐사법의 수행 과정(2)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]
19	1	1	3		5			8										18
19	1	1	3		5			8	9									18
19	1	1	3		5			8	9					14				18
19	1	1	3		5		7	8	9					14				18
19	1	1	3		5	5	7	8	9					14				18
19	1	1	3		5	5	7	8	9	24				14				18
19	1	1	3	1	5	5	7	8	9	24				14				18
19	1	1	3	1	5	5	7	8	9	24			13	14				18
19	1	1	3	1	5	5	7	8	9	24		12	13	14		16		18
19	1	1	3	1	5	5	7	8	9	24	5	12	13	14		16		18

성능 특성

- ◆ 해시 테이블이 2/3 정도 차 있을 경우, 선형 탐사법은 평균적으로 5번보다 적은 탐사를 수행
- ◆ 성공적인 탐색은 테이블이 90% 정도 찼을 때까지 5번 이하의 탐사로 가능
- ◆ 성공적이지 않은 탐색은 항상 성공적인 탐색에 비해 비용이 많이 듦

이중 해싱(double hashing)

◆ 클러스터링 문제를 해결

- 선형 탐사법은 두 번째 이후에 탐사되는 위치는 초기 탐사 위치에 따라 결정
- 두 번째 이후 탐사되는 위치가 첫 번째 탐사 위치와 무관하다면 클러스터링 문제를 완화시킬 수 있음

해시 함수의 예

$$h_1(k): k \bmod 19$$

$$h_2(k): 8 - (k \bmod 8)$$

키 :	1	19	5	1	18	3	8	9	14	7	5	24	1	13	16	12	5
-----	---	----	---	---	----	---	---	---	----	---	---	----	---	----	----	----	---

해시 1 : 1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5

해시 2 : 7 5 3 7 6 5 8 7 2 1 3 8 7 3 8 4 3

이중 해싱의 수행과정

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]
	1																	
19	1																	
19	1				5													
19	1				5			1										
19	1				5			1										18
19	1		3		5			1										18
19	1		3		5			1								8		18
19	1		3		5			1	9							8		18
19	1		3		5			1	9					14		8		18
19	1		3		5		7	1	9					14		8		18
19	1		3		5		7	1	9		5			14		8		18
19	1		3		5		7	1	9		5		24	14		8		18
19	1		3		5		7	1	9		5		24	14	1	8		18
19	1		3		5	13	7	1	9		5		24	14	1	8		18
19	1	16	3		5	13	7	1	9		5		24	14	1	8		18

성능 특성

- ◆ 이중 해싱은 평균적으로 선형 탐사보다 적은 탐사 회수를 가짐
- ◆ 이중 해싱은 선형 탐사에 비해 작은 테이블 크기를 가지고도 동일한 탐색 성능을 나타냄
- ◆ 테이블이 80% 이하로 채워져 있을 경우 성공적이지 않은 탐색의 평균 탐사 회수는 5번 이하이며, 99% 채워져 있을 경우 성공적인 탐색을 5번 이하로 할 수 있음

해싱과 이진 트리

- ◆ 이진 트리보다 해싱을 많이 사용하는 이유
 - 간단함
 - 상대적으로 빠른 탐색 시간
- ◆ 이진 트리의 장점
 - 동적 (삽입 회수를 미리 알고 있지 않아도 됨)
 - 최악의 경우 성능을 보장 (해싱의 경우 아무리 좋은 해시 함수라도 모든 값을 같은 장소로 해싱하는 경우가 발생)
 - 사용할 수 있는 연산의 종류가 많음 (예 : 정렬 연산)

기수 탐색(radix searching)

- ◆ 탐색 키의 디지털 성질(0과 1)을 이용해 탐색을 위한 이진 트리를 만들어 탐색을 진행
- ◆ 탐색 키의 해당 비트가 0이면 왼쪽 자식을 방문하고, 1이면 오른쪽 자식을 방문
- ◆ 탐색 키가 클 때는 노드를 방문할 때마다 탐색 키를 비교하는 것이 성능에 많은 영향을 줌
- ◆ 기수 탐색 알고리즘은 탐색 키의 비교 횟수를 줄이면서 기억 장소의 낭비를 줄이고 구현을 쉽게 하는 방향으로 개발되어 옴

1~26까지의 이진수

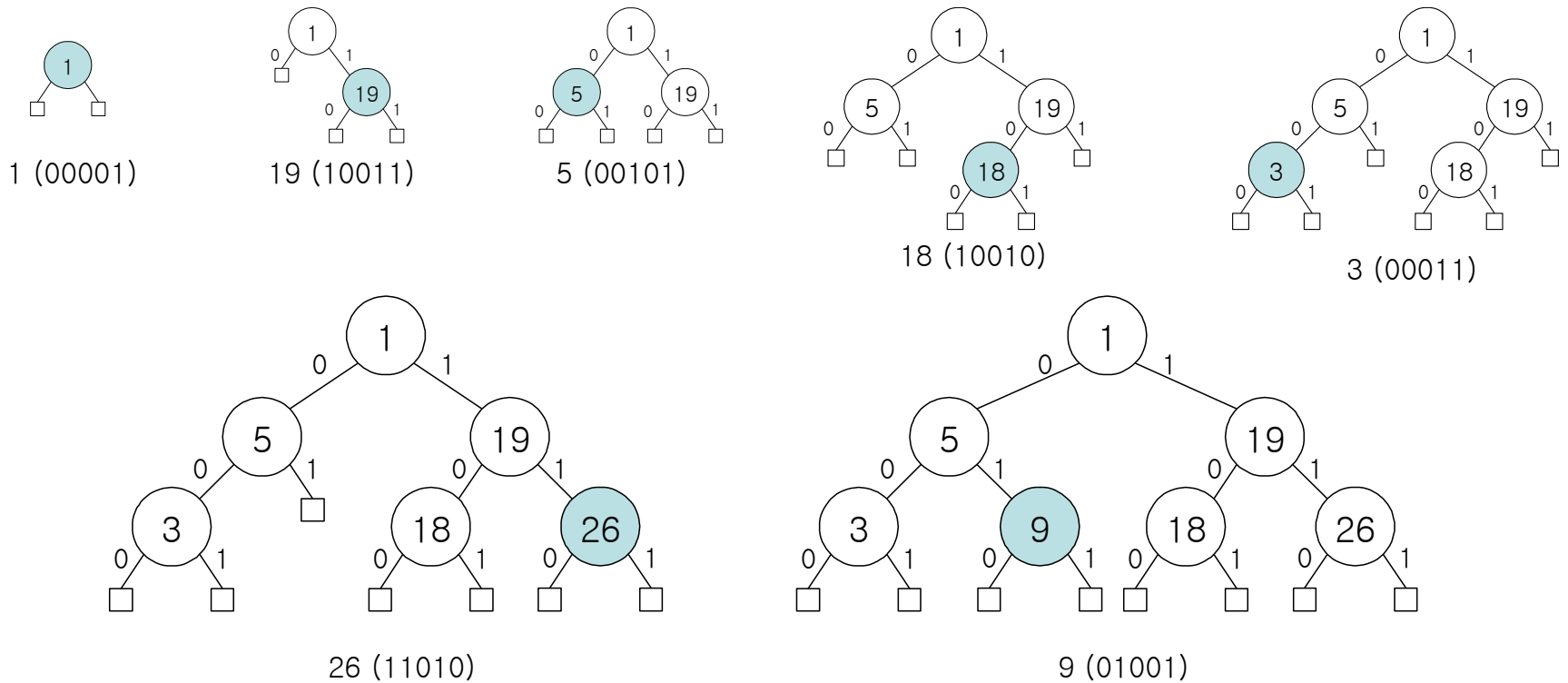
1: 00001	2: 00010	3: 00011	4: 00100
5: 00101	6: 00110	7: 00111	8: 01000
9: 01001	10: 01010	11: 01011	12: 01100
13: 01101	14: 01110	15: 01111	16: 10000
17: 10001	18: 10010	19: 10011	20: 10100
21: 10101	22: 10110	23: 10111	24: 11000
25: 11001	26: 11010		

디지털 탐색 트리(digital search tree)

- ◆ 탐색 키의 해당 비트가 0이면 왼쪽, 1이면 오른쪽으로 진행하여 단말 노드까지 이르면 그 곳에 새로운 노드를 삽입
- ◆ 노드를 방문할 때마다 탐색 키를 비교해야 하므로 탐색 키가 큰 경우 키 비교에 많은 시간이 걸림

디지털 탐색 트리의 구축 과정

◆ 키 (1, 19, 5, 18, 3, 26, 9)



성능 특성

◆ 장점

- 이해하기 쉽고 구현이 간단함

◆ 단점

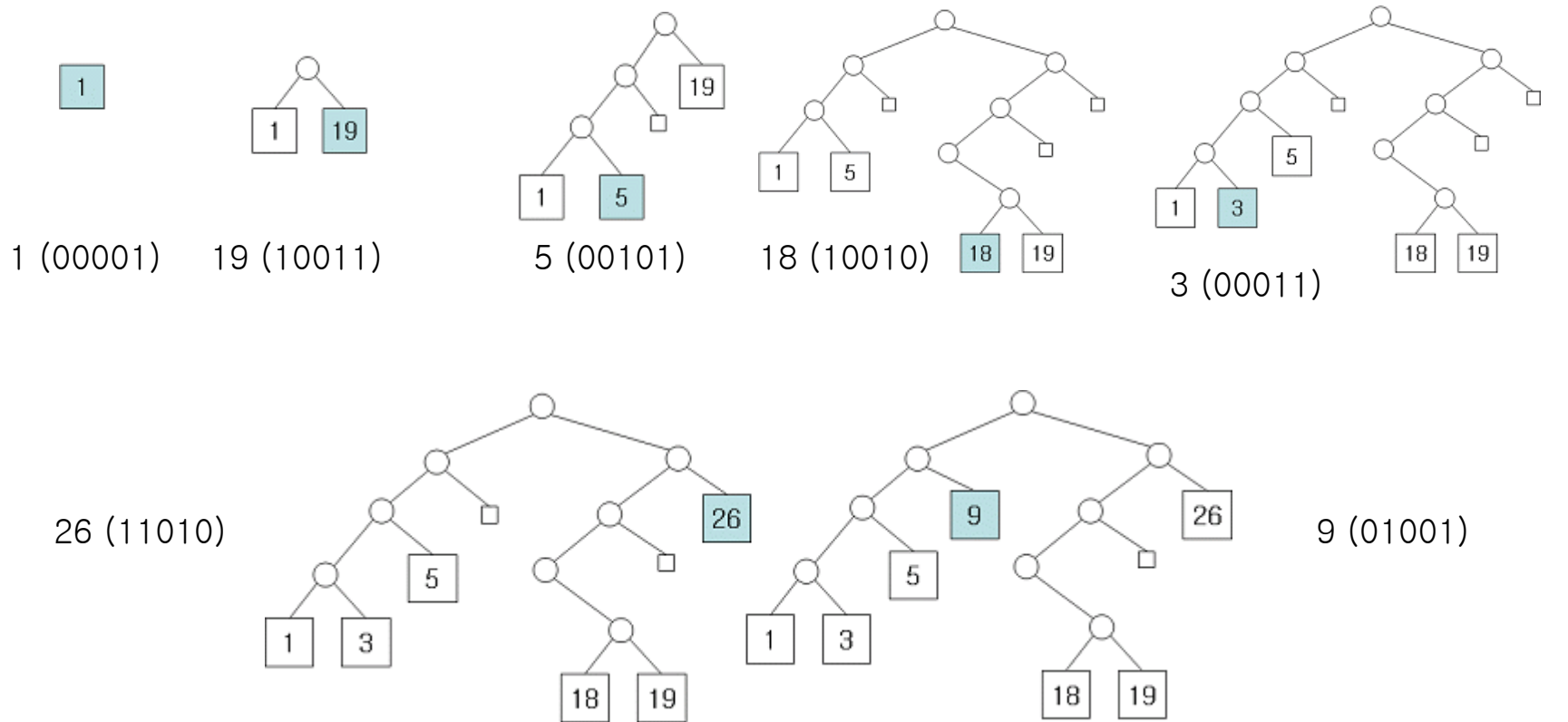
- 탐색 키의 비트에 따라 노드를 방문 할 때마다 디지털 탐색 트리의 키와 탐색 키를 매번 비교해야 하므로, 탐색 키가 큰 경우에는 키 비교에 많은 시간이 소요

기수 탐색 트라이(radix search trie)

- ◆ 트라이(trie) : reTRIEval 에 유용하다고 하여 Fredkin이 명명함
- ◆ 노드를 내부 노드와 외부 노드로 나누어 내부 노드는 탐색시 왼쪽과 오른쪽 이동만을 나타내고 키는 외부 노드에만 저장
- ◆ 탐색 키 비교는 외부 노드에서만 이루어지므로 탐색당 1번만 비교함
- ◆ 내부 노드는 키를 저장하지 않으므로 기억 장소의 낭비가 심하고, 내부 노드와 외부 노드를 나누어 프로그래밍해야 하므로 프로그래밍하기가 어려움

기수 탐색 트라이의 구축 과정

◆ 키 (1, 19, 5, 18, 3, 26, 9)



성능 특성

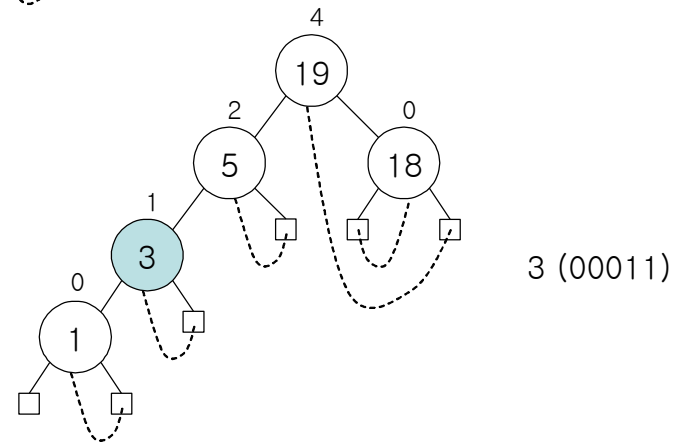
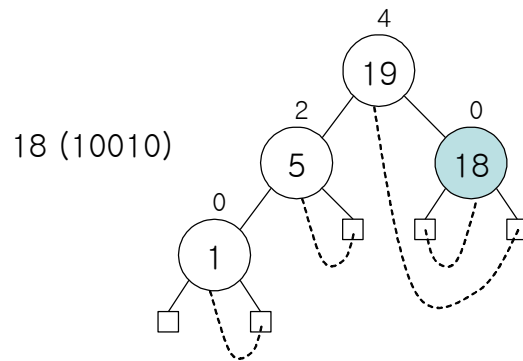
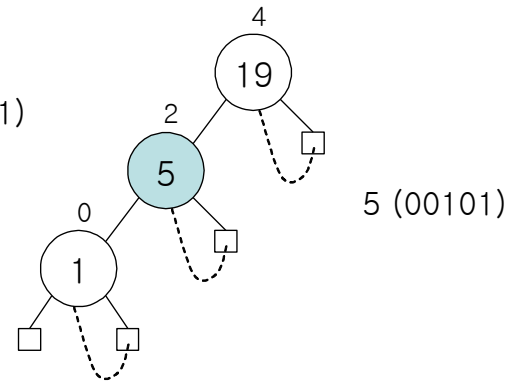
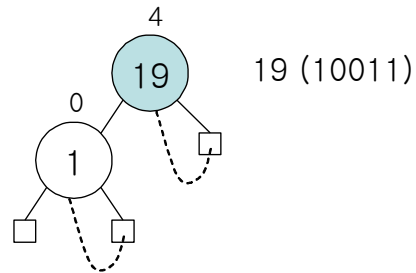
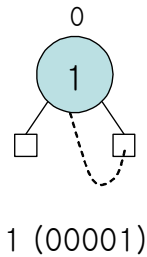
- ◆ 탐색 당 한 번의 키 비교만을 수행하므로 비교 시간을 절약
- ◆ 내부 노드에는 키를 저장하지 않으므로 기억 장소의 낭비 심함
- ◆ 내부 노드와 외부 노드를 나누어 관리해야 하므로 프로그래밍하기가 어려움

패트리샤 트리(patricia tree)

- ◆ “Practical Algorithm To Retrieve Information Coded In Alphanumeric”의 약자
- ◆ 노드에 몇 번째 비트를 비교할 것인지를 나타내는 숫자를 통해 내부 노드와 외부 노드의 구분을 없애서 기억 장소를 절약
- ◆ 위쪽 링크(upward link)를 만나면 위쪽 링크가 가리키는 노드와 탐색 키를 비교하게 되므로 탐색당 1번만 비교 수행
- ◆ 노드마다 키를 비교해야 하는 디지털 탐색 트리의 단점과 내부 노드와 외부 노드를 두어 기억 장소를 낭비하는 기수 탐색 트라이의 단점을 동시에 극복한 방법

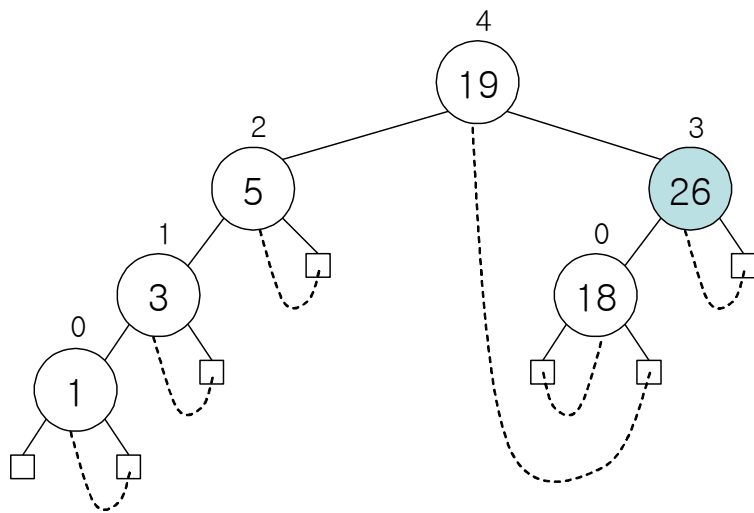
패트리샤 트리의 구축 과정(1)

◆ 키 (1, 19, 5, 18, 3, 26, 9)

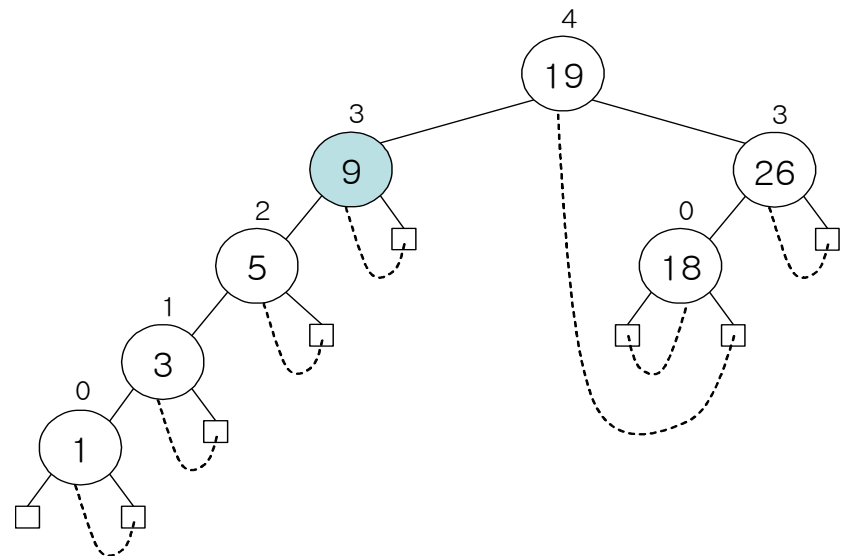


패트리샤 트리의 구축 과정(2)

◆ 키 (1, 19, 5, 18, 3, 26, 9)



26 (11010)



9 (01001)

성능 특성

- ◆ 내부 노드에 키를 저장하여 기억 장소를 절약
- ◆ 상향 링크를 만날 때만 키 비교를 수행하게 하여 탐색 당 한 번의 비교만을 수행
- ◆ 디지털 탐색 트리와 기수 탐색 트라이의 단점을 동시에 극복

외부 탐색(external searching)

- ◆ 매우 큰 화일에 있는 데이터에 빠르게 접근하는 것은 많은 응용에서 매우 중요함
- ◆ 외부 탐색은 큰 디스크 화일을 주로 다룸
 - 테이프 같은 순차 접근 장치는 순차 탐색 외에 별다른 탐색 방법이 없음
- ◆ 외부 탐색 기법은 이미 배운 내부 탐색 기법의 논리적 확장임
- ◆ 10억개 이상의 데이터를 탐색하는데 불과 2~3회의 디스크 접근으로 가능함
- ◆ 디스크는 페이지로 구분되어 있으며, 페이지는 많은 레코드를 가지고 있음

인덱스된 순차 접근

- ◆ 탐색 속도를 높이기 위해 탐색 키가 디스크 상의 어떤 페이지에 있는지 가리키는 인덱스를 유지
- ◆ 인덱스 페이지는 데이터 페이지보다 많은 키와 페이지 인덱스 저장 가능
- ◆ 주기억 장치에 마스터 인덱스 유지
 - 마스터 인덱스 : 디스크에 저장된 키 값에 대한 정보
- ◆ 인덱스된 순차 화일에서 탐색은 평균 2번 이내의 디스크 접근을 필요로 하지만, 삽입은 전체 화일을 재구성해야 하는 단점이 있음

레코드 저장 예

◆ 키

- 5,24,20,5,18,14,1,12,19,5,1,18,3,8,9,14,7,5,24,1,1
3,16,12,5

◆ 순차 접근

	페이지 0	페이지 1	페이지 2
디스크 1	1 1 1 3 ■	5 5 5 5 ■	5 7 8 9 ■
디스크 2	12 12 13 14 ■	14 16 18 18 ■	19 20 24 24 ■

◆ 인덱스된 순차 접근

	페이지 0	페이지 1	페이지 2
디스크 1	1 3 5 1 2 + ■	1 1 1 3 ■	5 5 5 5 ■
디스크 2	5 9 14 1 2 + ■	5 7 8 9 ■	12 12 13 14 ■
디스크 3	14 18 24 1 2 + ■	14 16 18 18 ■	19 20 24 24 ■

성능 특성

- ◆ 탐색은 평균 2번 이내의 디스크 접근 필요
- ◆ 삽입은 전체 화일을 재구성해야 하는 단점

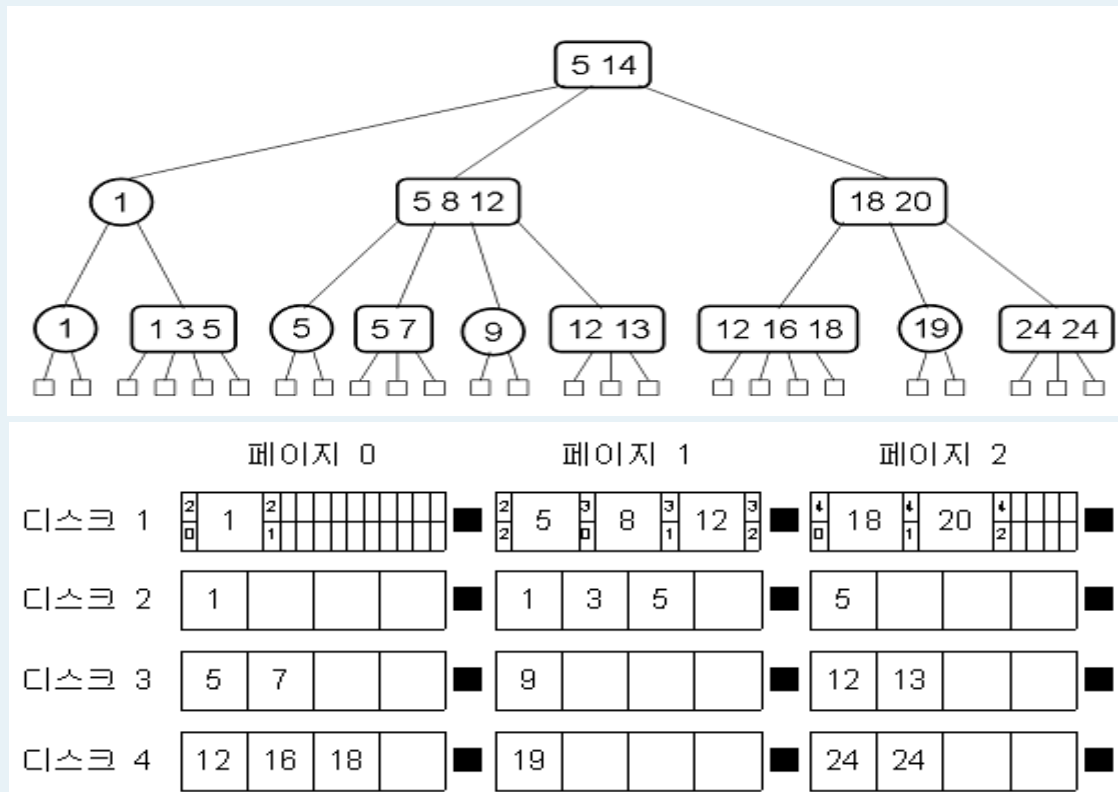
B-트리

- ◆ R. Bayer와 E. McCreight는 다방향 균형 트리를 외부 탐색에 최초로 사용함
- ◆ 균형 트리로서 각 노드가 $M-1$ 개까지의 키 값을 가지고 M 개의 포인터를 가짐
- ◆ 차수가 M 인 B-트리의 각 노드는 $M/2$ 개 이상의 키 값을 가지도록 유지됨
- ◆ 트리를 내려 가다가 가득 찬 노드를 만나면 분할함
 - 하나의 M -노드가 연결된 k -노드는 두 개의 $(M/2)$ -노드가 연결된 $(k+1)$ -노드로 분할됨
- ◆ 루트 노드는 주기억 장소에 저장

B-트리 접근의 예

◆ 키

- 5, 24, 20, 5, 18, 14, 1, 12, 19, 5, 1, 18, 3, 8, 9, 14, 7, 5, 24, 1, 13, 16, 12, 5



성능 특성

- ◆ N 개의 데이터 레코드를 가지고 차수가 M 인 B-트리에서 주어진 키 값으로 탐색하거나 삽입할 때 필요한 디스크 접근 회수는 $\log_{M/2} N$ 보다 작음
- ◆ 이 값은 N 개의 데이터 레코드를 가지고 모든 노드가 $M/2$ 개의 키 값을 가진 B-트리를 구성했을 때의 높이에 해당함

B-트리의 외부노드로 구성

- ◆ 내부노드에는 전체 레코드가 아닌 키 값만을 저장
 - ⇒ 상위 레벨에서 더 큰 M 값을 가질 수 있음
- ◆ 두 가지 분기 요소
 - M_I : 내부노드에서 둘이나 네 페이지에 저장할 수 있는 레코드의 수
 - M_B : 외부노드에서 한 페이지에 저장할 수 있는 레코드의 수
- ◆ 예 : $M_I=2048$ 이고 3레벨을 가진 B-트리는 수십억 개(1024^3)이상의 원소를 다룰 수 있음

B-트리의 외부노드로 구성의 예

◆ $M_I = 8, M_B = 5$

